# A Heuristic Approach to Algorithm-Completion in the ADHOC Environment

Author:
Kenneth Harvey
kh2333@columbia.edu

Advisor:
Alfred Aho
aho@cs.columbia.edu

## I. Abstract

*This paper discusses some motivations for the use of heuristic (rather than statistical) approaches to algorithm completion in programming tools. In particular, it stresses the difference between algorithms and programs, it analyzes the particular algorithm completion methods used in the ADHOC programming environment, and it draws comparisons to existing completion methods, which are largely statistics-based. Finally this paper explores some avenues for future work in algorithm completion. This paper is intended largely as a survey of (and commentary on) existing code and algorithm completion technologies.*

**Keywords**: algorithms, code completion, software engineering

## II. Motivations

There is an exponentially increasing amount of program code written each year [4], as well as a swiftly increasing number of languages in which program code is written. Fundamentally, however, this code expresses a far smaller (and slower-growing) set of algorithms and actions that programmers have devised to accomplish particular tasks. This means that a lot of code that gets written is algorithmically similar (even identical) to code that already exists. It makes sense when we think about, say, an encryption algorithm being ported to a new target language, or two web applications that target the same REST API processing their results in similar ways. However, these kinds of algorithmic duplicates often cost just as many person-hours to implement as did their original implementations, and they are susceptible to many of the well-studied maintainability problems of "code clones" [7].

There are several benefits that can be attained from identifying and reducing algorithmic duplication. As mentioned, it saves person-hours. If we can help a developer identify and reference an existing algorithmic solution, implementing that solution in code will be much quicker than producing *and* implementing a new solution from scratch. Second, it improves software maintainability and correctness. By promoting encapsulation we reduce the overall project interdependencies [12]. Third, it promotes reusable library functions and language features. By identifying common duplicates, we can generate a list of most-sought (but currently lacking) functionality. Finally, it makes programming easier for novices. By suggesting completions for some of the more rote programming routines, we can help beginners focus more on the logic they are trying to code.

This paper's main contributions are a discussion of the merits of using algorithms rather than programs for detecting similarity, an analysis of a practical heuristic-based approach taken in the ADHOC environment [5], and a commentary on existing statistical methods for code completion.

## III. Algorithms vs Programs

Though variations of these definitions can be discussed [8], this paper will use the word "algorithm" to refer to an abstract solution to a formally described problem, and use the word "program" to refer to a specific implementation of one or more such solutions that is targeted at a particular execution environment. The job of a "programmer", then, is first to understand a problem's description, then to devise an algorithm that solves the problem, and finally to express the algorithm in terms of a program for the target environment. Brooks [3] refers to these three steps respectively as "understanding", "method-finding", and "coding".

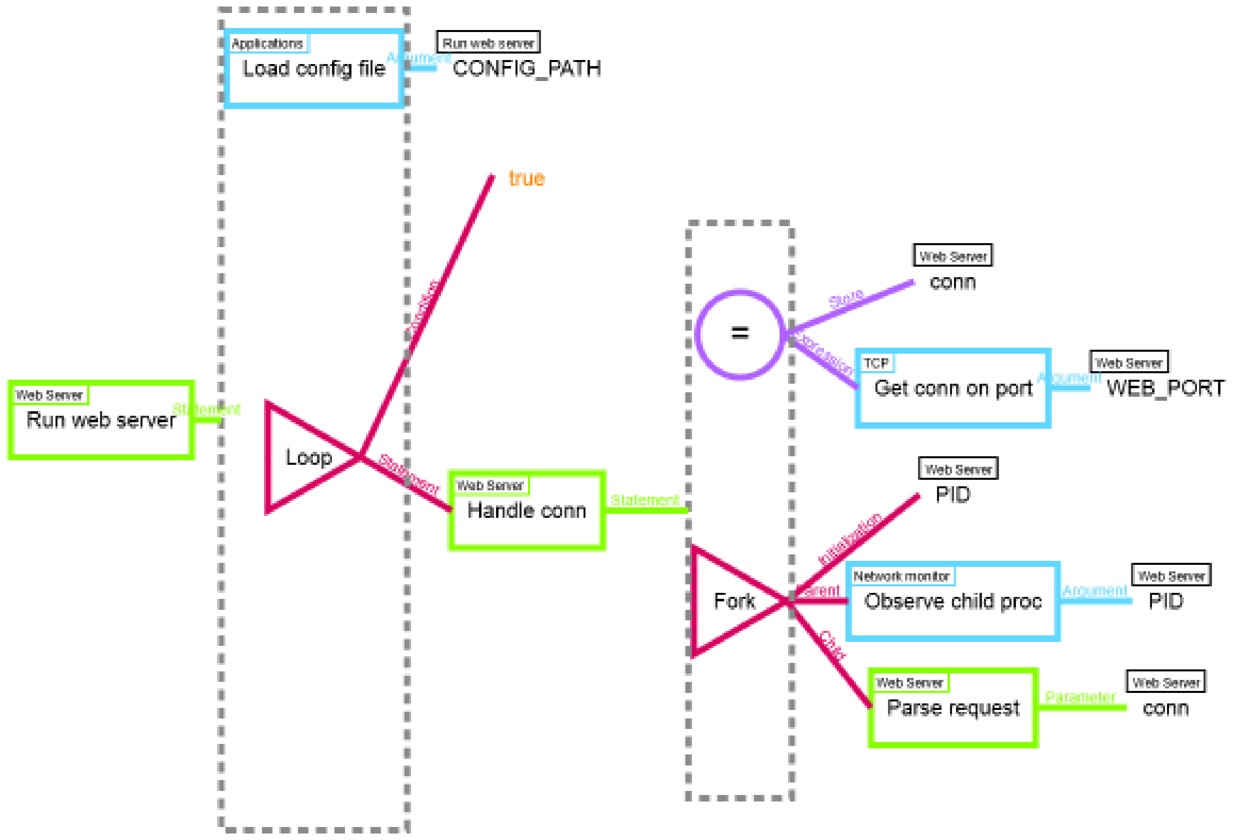
Understanding and method-finding are topics that reach into cognition and psychology, and they can often be analyzed at a biological level [10] more easily than in a computer science setting. By contrast, the cognitive processes involved in coding are very mechanical, and often include the kinds of rote look-up and repetition tasks that computers are great at doing for us. Going back to the example of an encryption algorithm being ported to a new language, the problem of encryption (presumably under some specific constraints) has already been understood and solved by the time the first implementation is written. So the port to the new language should only consist of mechanical re-coding.

Intuition suggests that this re-coding should be mostly automatic, and that software tools should do most of the heavy lifting for us, however, in practice, this is not often the case. Enter algorithm completion.

## IV. Algorithm-Completion in the ADHOC Environment

Before discussing algorithm-completion, however, it will be helpful to provide a brief overview of the ADHOC project [5]. ADHOC (the Action-Driven Human-Oriented Compiler) is a visual programming environment, designed to keep a programmer's focus on the structures of algorithms rather than on their implementations. It attempts this by obeying an "Action-Oriented" paradigm, in which the entire algorithm is described in terms of component "actions", which are then viewed as algorithms in their own right. These sub-algorithms are composed of still more actions, and so on, and the result is a graphical tree structure describing the entire original algorithm. This contrasts with the Object-Oriented paradigm, in which everything is an object, or a method invoked upon objects. AOP behaves more like functional programming with side-effects.

An example of ADHOC usage is reproduced in Figure 1 (next page). Rectangular, solid-bordered nodes represent actions, with the green ones being defined and immediately used in the current algorithm, and the blue/teal ones being defined in another "package", but used in the current algorithm. ADHOC can be used as a planning tool for designing high-level systems. It allows for some component actions to be described later, for example the "Parse request" action in Figure 1. It can also be used to describe lower level arithmetic and data manipulation. In either case, once the programmer manually describes the algorithm, the ADHOC system will perform some preliminary validation, and then automatically generate executable code in a given target language (the C code generated from the example in Figure 1 is presented in Appendix A). ADHOC currently supports a subset of C and a subset of Javascript for code generation, but more languages and features are forthcoming. More detail on ADHOC's usage and other information can be found on the project homepage [5].



*(Figure 1) An ADHOC action describing an algorithm to run a web server. Note that the final "Parse request" action has not yet been described*

Given ADHOC's focus on algorithm structure, it is not surprising that it employs an algorithm-completion mechanism rather than a program-completion one. Here is how ADHOC's algorithm-completion works:

1) The programmer creates an action node, and begins describing its structure (by adding child nodes that are either parameters or statements).

2) After providing a partial description, the programmer selects the action node, and requests an analysis of it (`[CTRL+a]`).

3) The ADHOC system performs a heuristic analysis of the selected action by looking at certain key features (e.g. whether it contains conditional return statements, what its return-type is, how many parameters it takes, etc.), and produces a feature vector of its findings. The feature vector is computed using a tree-walk of the selected action's sub-tree, so it is generated in time linear to the number of nodes in that sub-tree. A full list of the features analyzed is provided in Appendix B.

4) The ADHOC system compares the feature vector of the selected action against a database of vectors from other known actions, and returns a list of the closest matches, along with comments and other meta-data about them. Closeness of match is determined by the following formula:

$$\delta_{s,k} = \sum_{i=1}^{F} \left| feat_{s,i} - feat_{k,i} \right| \cdot w_i$$

(In other words, the *difference rating* between the selected action *s* and a known action *k* is the sum of the difference between feature *i* in *s* and feature *i* in *k* times the weight of feature *i* for each of the F features listed in Appendix B)

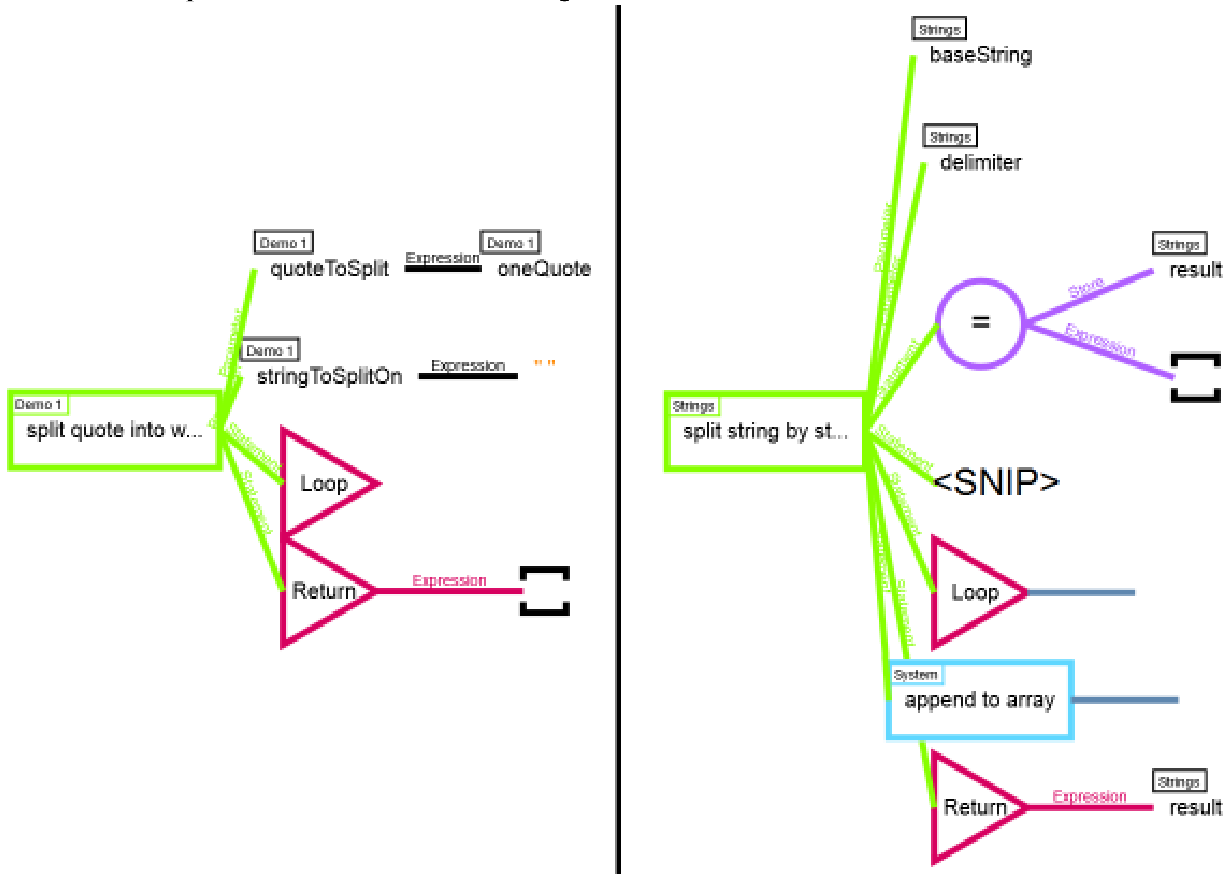
5) The programmer views the list of matched actions, and selects the one that is the best fit for the algorithm s/he is describing.

6) The ADHOC system fetches a serialization of the best-fit action's node structure and displays it to the programmer for confirmation.

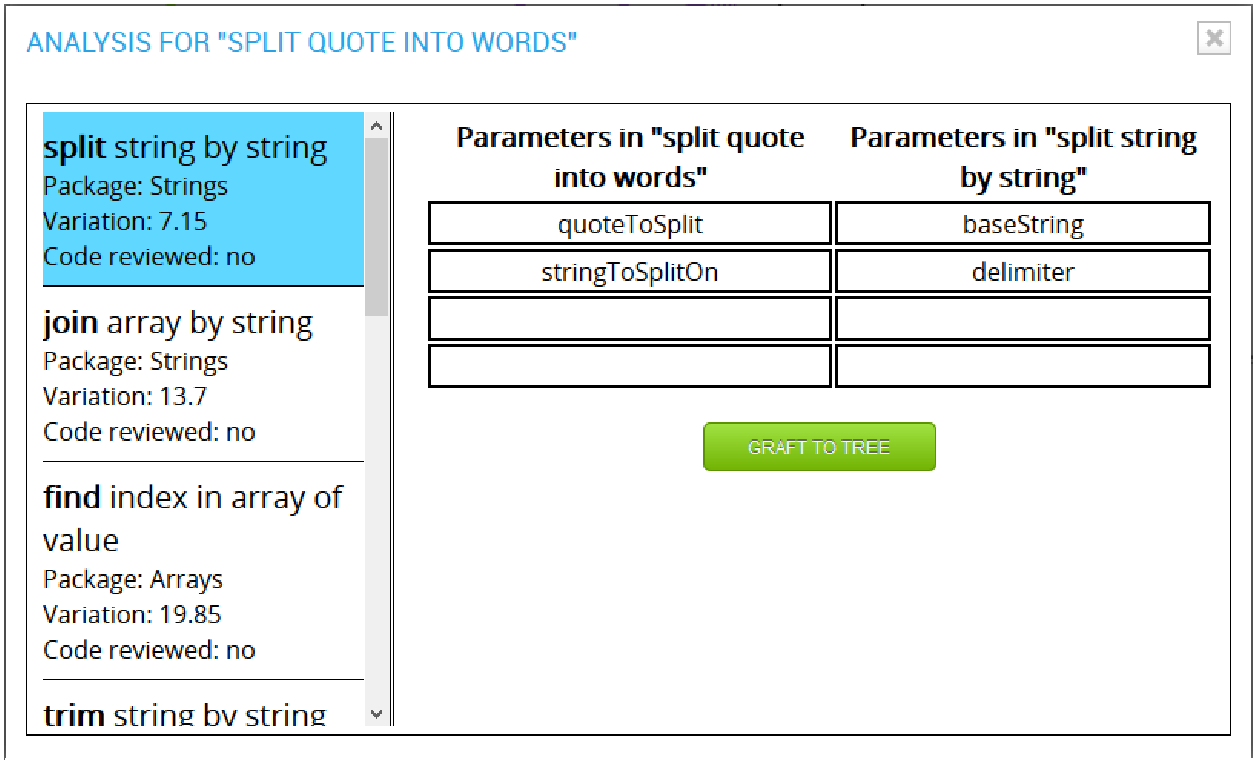
7) The programmer matches any existing parameters from his/her partial sub-tree with the corresponding parameters in the best-fit action

8) The ADHOC system splices the best-fit action's sub-tree over the programmer's partial sub-tree, maintaining any parameter names the programmer was using, and smoothing/confirming any datatypes that may need to be altered.

Figure 2 (below) displays a contrived example. On the left is the start of an algorithm that will split a quotation into an array of words. On the right is a finished description of another algorithm which splits an arbitrary string by some delimiter. It's no surprise that these two algorithms would be heuristically similar. Even though the left action is only partially complete, the two only have a differrence rating of 7.15, while two entirely dissimilar actions will generally have a rating around 50 or 60 (over 100 in extreme cases). Figure 3 (next page) displays the interface ADHOC provides for the programmer to confirm splicing the fetched action onto the



*(Figure 2) Left: The start of an action that will split a quotation into words. Right: A completed action that splits an arbitrary string by some delimiter (some variable assignments were snipped and some sub-trees folded to save space).*


ANALYSIS FOR "SPLIT QUOTE INTO WORDS"

**split** string by string
Package: Strings
Variation: 7.15
Code reviewed: no

**join** array by string
Package: Strings
Variation: 13.7
Code reviewed: no

**find** index in array of value
Package: Arrays
Variation: 19.85
Code reviewed: no

**trim** string by string

| Parameters in "split quote into words" | Parameters in "split string by string" |
|---|---|
| quoteToSplit | baseString |
| stringToSplitOn | delimiter |
| | |
| | |

*(Figure 3) Choosing parameters to preserve from the selected action to the fetched known action.*

existing action. In the left column are actions deemed similar to "split quote into words".

A less obvious example (not shown here, due to its size, but available upon request), is that of a subsequent action, "print quote info", which was intended to compute and print the most frequent word and the average word length from the array of words returned by "split quote into words". ADHOC found "print quote info" to be most similar to an action in the Math package for computing the mean, median, and mode of a list of numbers. These actions had a difference rating of 15.15, which is still very low, considering that one action had not been completed, and that they were intended for different purposes. Once spliced, only two modifications had to be added to the "mean median mode" action: one (due to computing the `average` of strings rather than numbers) was to sum up the `size()` of each element in the input array rather than its value, and another to remove the `mode` computation completely. Making these two modifications took less than a minute and was far easier than figuring out the full correct description of "print quote info". Upon examination, it is clear why these two algorithms would have such similar descriptions, however, the relationship might not be obvious to a programmer tasked with implementing one of them, without the context of the other.

## V. Comparison to Stat. Approaches

Code similarity analysis techniques have existed for some time [7], and their application for code completion tools seems to be a growing trend in software engineering [2][9][13]. As ADHOC does, so do many of the existing techniques try to analyze whole algorithms, and to look beyond trivial syntactic differences. However, unlike ADHOC, most of the other techniques use textual source-code analysis, and so they generally opt for statistical (rather than heuristic) approaches for recognizing similarity. Unfortunately, statistics introduce syntax dependence. These systems have to be retrained on new languages' syntax, so they have trouble detecting algorithmic similarity across languages. Also, many of them use n-grams or other immediate-vicinity text patterns, so they cannot analyze a large action in depth. Furthermore, statistical approaches are inherently limited to combinations of tokens that have been seen together before. It is possible for descriptively similar algorithms to be implemented with substantially different semantic tokens.

The following is a quick look at the state-of-the-art in code completion today.

**SLANG** [9] is an API-based system that trains on "snippets" of code found on public repositories. APIs provide a substantial amount of semantic rigidity, so this is a sensible starting point. SLANG focuses on method calls (similar to ADHOC's focus on actions), and it uses facets such as a value's position in the method call's argument list to determine the most common textual match. Once the best match is found, SLANG performs some validation to smooth the fetched snippet into the surrounding code.

**Bing Developer Assistant** [13] is also API-based, and also treats "snippets" as the primary units of code completion. Unlike SLANG, however, it has the advantage of being hooked into Visual Studio's IntelliSense framework, giving it access to information about variables in scope, object fields, class constants, etc.

**PLINY** [2] is still in the very early stages of its development, however, its promotional video promises superior semantic analysis of samples from large public code-bases. Like ADHOC, PLINY relies on a database of features vectors tied to code examples in order to match the semantic meaning of the programmer's code. In particular, PLINY analyses "interesting features, like functions, variables, [and] dependencies". Like SLANG and ADHOC, PLINY smooths the retrieved code to match the surrounding program.

**Code Clone Detection** is a field with almost the opposite mentality from that of code completion. Instead of encouraging programmers to copy existing code or algorithms, it focuses on identifying such copies for removal or refactoring (more on this in Section VI below). Nonetheless, there are parallels between the ways in which clones are detected and the ways in which code is scanned for completion matching. Some of the top methods of code clone detection are:

- *AST sub-tree comparison using hashing and root-node type* [1]: This is performed by ADHOC at the algorithm level, but it is still effectively looking at a parse tree.

- *AST sub-tree comparison using fingerprinting* [11]: ADHOC uses a heuristic fingerprint, but others use statistical ones.

- *"Characteristic Vector" comparison* [6]: Jiang et al.'s DECKARD system uses Hamming and Euclidian distances on AST node type count vectors (ADHOC includes more characteristic features, but only looks at the Hamming distance).

## VI. Avenues for Future Work

As mentioned in Section II, code duplication is a big problem, especially in large systems with many developers. Code completion tools can help prevent unwitting duplication of algorithmic work, but they exacerbate the problem of code clones. One hope is that the same methods for locating and copying existing code can also be used to alert language and system developers to the need for new functionality in their platforms. The next iterations of ADHOC will allow known algorithms to be referenced instead of being spliced directly into the project. In this way known algorithms can become like library functions which are shared across projects, at least until the target code is generated.

There are also interesting things that can be done during the heuristic analysis. Due to time-constraints, the ADHOC feature vectors were kept relatively simple (see Appendix B), but much richer and more nuanced feature vectors could be used. In particular, drawing on some code clone detection techniques [11], features could be added that represent the inclusion of less common operators and actions, as those tend to be characteristic of certain classes of algorithms. It would also be interesting to apply machine learning techniques to the heuristic process to find more accurate weights for the features (the current weights are mostly best guesses).

Finally there's plenty of work to be done on the ADHOC project in general. There are plenty of UI issues to clean up, some bugs to fix in the generated code. On top of that, support for more languages and features is a must! In particular, generating LLVM IR from ADHOC algorithm descriptions should prove fruitful for a number of applications. Finally, it would be helpful to create parsers for some existing languages that output in ADHOC's `.adh` format. This would allow in-file static analysis, code clone detection, and validation, as well as language-to-language translation.

With such opportunities available, heuristic algorithm analysis is a topic to watch.

## VII. Sources

[1] Baxter I D, et al. Clone Detection Using Abstract Syntax Trees. ICSM, 1998.

[2] Boyd J. Next for DARPA: Autocomplete for programmers. Rice Univ News, 2014.

[3] Brooks R. Towards a theory of the cognitive processes in computer programming. Int J of Man-Machine Studies, 1977. 51(2): p197-211.

[4] Deshpande A, Riehle D. The Total Growth of Open Source. Proc Conf Open Src Sys, 2008.

[5] Harvey, K. ADHOC. HarveyServ, 2014.

[6] Jiang L, et al. DECKARD Scalable and Accurate Tree-based Detection of Code Clones. Proc Int Conf Software Eng, 2007.

[7] Lague B, et al. Assessing the benefits of incorporating function clone detection in a development process. ICSM, 1997.

[8] Rachit. When are two algorithms said to be "similar"? Theor Comp Sci - Stack Exchange, 2012.

[9] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. PLDI, 2014.

[10] Siegmund J K, et al. Understanding Understanding Source Code with Functional Magnetic Resonance Imaging. Comm of the ACM, 2014.

[11] Smith R, Horwitz S. Detecting and Measuring Similarity in Code Clones. IWSC, 2009.

[12] Snyder A. Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not, 1986.

[13] Team, B D C. Bing Developer Assistant for Visual Studio focuses on improving productivity within the experience. Bing Dev Center Team Blog, 2014

## VIII. Appendices

### A. C code generated from the example in Figure 1

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <libadhoc.h>
#include "Applications.h"
#include "TCP.h"
#include "Network_monitor.h"

// Declare global variables
adhoc_data* CONFIG_PATH;
int WEB_PORT;

void Run_web_server();

void Handle_conn();

void Parse_request(adhoc_data* conn);


void Run_web_server(){

    // Body of Run_web_server
    Load_config_file(CONFIG_PATH);
    for(; true; ){
        Handle_conn();
    }
}

void Handle_conn(){
    // Declare variables in scope
    adhoc_data* conn = NULL;
    void PID = NULL;

    // Body of Handle_conn
    conn = adhoc_referenceData(Get_conn_on_port(WEB_PORT));

    // -- FORKING IS NOT YET IMPLEMENTED -

    // Reduce references on complex vars in scope
    adhoc_unreferenceData(conn);

}

void Parse_request(adhoc_data* conn){
    adhoc_referenceData(conn);

    // Body of Parse_request

    adhoc_unreferenceData(conn);
}

// Main function for execution
int main(int argc, char **argv){
    // Initialize global vars
    CONFIG_PATH = adhoc_referenceData(adhoc_createString("path/to/config"));
    WEB_PORT = 80;
    Run_web_server();

    // Reduce references on complex global vars
    adhoc_unreferenceData(CONFIG_PATH);

    return 0;
}
```

## B. List of heuristic features currently employed by ADHOC, and associated weights

| **Feature** | **Wieght** | **Description** |
|---|---|---|
| `totalLoops` | `4.0` | (Integer) The total number of loops found in the action's sub-tree, including in called actions |
| `maxLoopNest` | `6.0` | (Integer) The deepest nesting of loops found in the action's sub-tree, including in called actions |
| `condReturns` | `3.0` | (Boolean 0,1) Represents whether or not the action can return without all descendent nodes being executed |
| `actionVerb` | `5.0` | (String) The first sequence of non-space characters in the action's name. Currently uses exact match, but will use M.E.D. |
| `nodeCount` | `0.05` | (Integer) Total number of nodes at any depth in the action's sub-tree |
| `paramCount` | `3.0` | (Integer) Total number of parameters to the action |
| `childCount` | `1.0` | (Integer) Total number of direct children (including parameters) of the action |
| `inputsVoid` | `0.5` | (Integer) Total number of void-typed parameters to the action (these will be illegal eventually) |
| `inputsBool` | `0.5` | (Integer) Total number of boolean-typed parameters to the action |
| `inputsInt` | `0.5` | (Integer) Total number of integer-typed parameters to the action |
| `inputsFloat` | `0.5` | (Integer) Total number of float-typed parameters to the action |
| `inputsString` | `0.5` | (Integer) Total number of string-typed parameters to the action |
| `inputsArray` | `0.5` | (Integer) Total number of array-typed parameters to the action |
| `inputsHash` | `0.5` | (Integer) Total number of hash-typed parameters to the action |
| `inputsStruct` | `0.5` | (Integer) Total number of struct-typed parameters to the action |
| `inputsAction` | `0.5` | (Integer) Total number of action-typed parameters to the action |
| `inputsMixed` | `0.5` | (Integer) Total number of mixed-typed parameters to the action |
| `outputType` | `3.0` | (Integer) The adhoc_dataType enumeration of the action's return type |